Objective C Programming

Lecture 3
Categories. Protocols. Frameworks. Foundation.

International University of Information Technology
Almaty, 2012

Ooops

- There was a typo in Lecture 2, "reduce" method in Fraction class

```
- (void) reduce {
    int u = numerator;
    int v = denominator;
    int temp;

    while (v!=0){
        temp = u%v;
        u=v;
        v=temp;
    }

    numerator /= u;
    denominator /= v;
}
```

```
- (void) reduce {
    int u = numerator;
    int v = denominator;
    int temp;

    while (v!=0){
        temp = u%v;
        u=v;
        v=temp;
    }

    numerator /= u;
    denominator /= u;
}
```

Categories

- Split a definition of class into groups
- Extend existing definition

Let's see an example in action

Categories

```
#import "ClassName.h"

@interface ClassName (CategoryName)
    // method definitions
@end

@implementation ClassName (CategoryName)
    // method implementation
@end
```

Categories

- Need to import Class definition first
- No limit on categories
- Class-Category name is unique

Class Extension

- A nameless category
- Allows to define new variables

Let's see an example in action

Class Extension

```
#import "ClassName.h"

@interface ClassName ()
    // variables definitions
    // method definitions
@end

@implementation ClassName
    // method implementation
@end
```

Class Extension

- Category can override a method in class. Please, don't do it! Original method won't be accessible.
- If need to override – create a subclass. Original method will be available via **`super`**

Protocols

- List of methods shared among classes
- Just a list. Methods might not be implemented.
- Some methods are required, some – optional\
- Implementing all required methods called **conforming** or **adopting** a protocol

Let's see an example in action

Protocols

```
@protocol ProtocolName
    // required methods
@optional
    // optional methods
@required
    // more required methods
@end

@interface ClassName : ParentClass <ProtocolName>
    //
@end

@implementation ClassName
    // methods implementation
@end
```

Protocols

- Doesn't reference any classes. Any class can conform to any protocol.
- Class can adopt several protocols

```
@interface ClassName : ParentClass <ProtocolName, AnotherProtocolName>
```

Checking conformance

```
if ([myObject conformsToProtocol: @protocol (ProtocolName)] == YES) {
    // can use methods declared in ProtocolName
}
```

Checking responsiveness

```
if ([myObject respondsToSelector: @selector (myMethod)] == YES){
    // can call myMethod on myObject
}
```

Checking conformance

```
id <myProtocol> myObject;
```

This tells the compiler that myObject will reference an object which conforms to myProtocol. If at some point it references an object that doesn't conform, compiler will issue a warning:

```
warning: class 'yourClass' does not implement the
'myProtocol' protocol
```

Protocols

- You can list several protocols

```
id <myProtocol, anotherOne> myObject;
```

-

-

Protocols

- You can list several protocols

```
id <myProtocol, anotherOne> myObject;
```

- You can extend existing protocols

```
@protocol anotherProtocol <myProtocol>
```

-

Protocols

- You can list several protocols

```
id <myProtocol, anotherOne> myObject;
```

- You can extend existing protocols

```
@protocol anotherProtocol <myProtocol>
```

- Categories can adopt protocols too

```
@interface Fraction (myCategory) <myProtocol, anotherOne>
```

Informal protocols

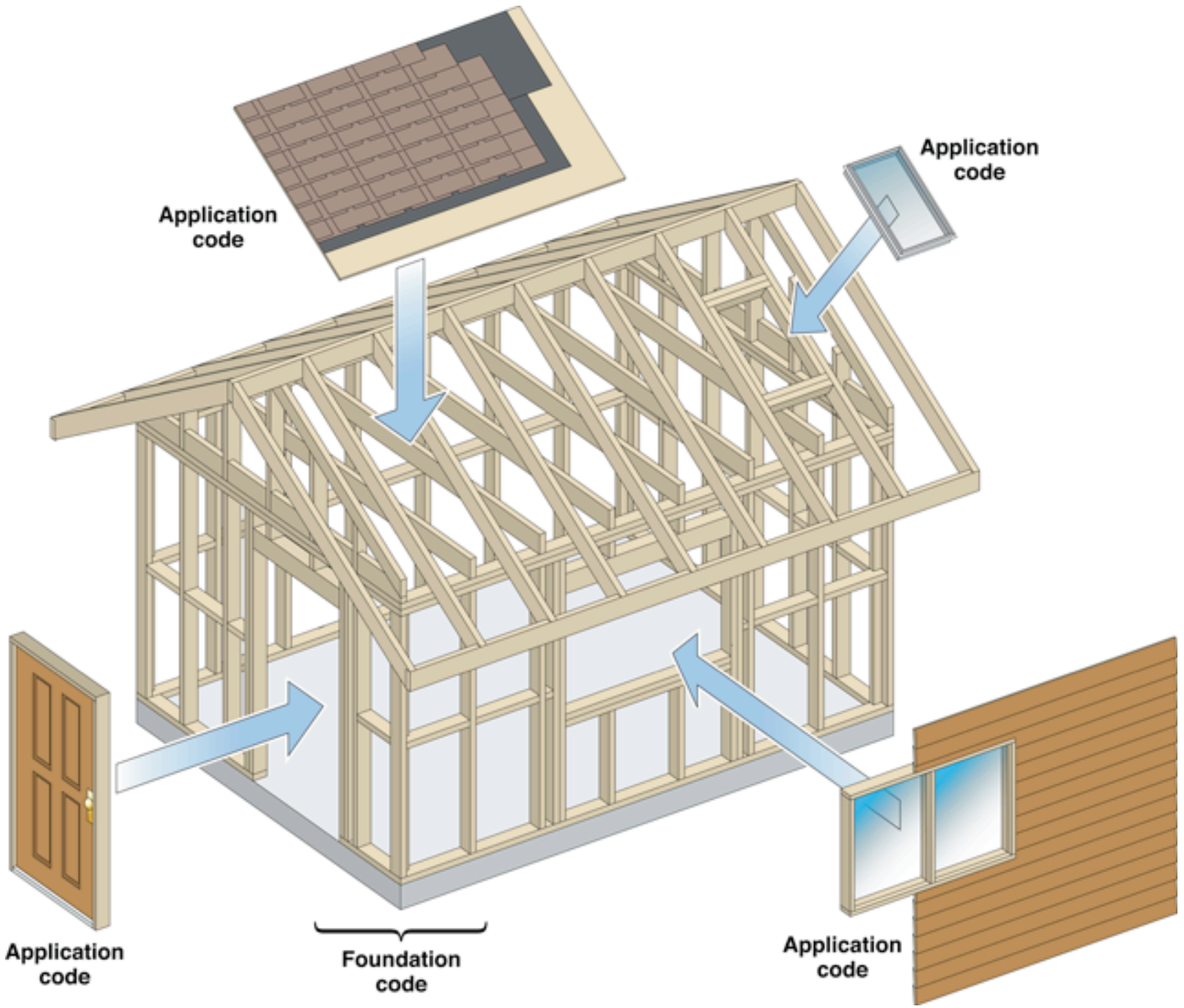
Just a category that lists some methods but doesn't implement them. This pattern is no longer used as @optional protocol methods are available in Objective C 2.0 and later

```
@interface NSObject (NSComparisonMethods)
- (BOOL)isEqualTo:(id)object;
- (BOOL)isLessThanOrEqualTo:(id)object;
- (BOOL)isLessThan:(id)object;
- (BOOL)isGreaterThanOrEqualTo:(id)object; - (BOOL)isGreaterThan:(id)object;
- (BOOL)isNotEqualTo:(id)object;
- (BOOL)doesContain:(id)object;
- (BOOL)isLike:(NSString *)object;
- (BOOL)isCaseInsensitiveLike:(NSString *)object;
@end
```

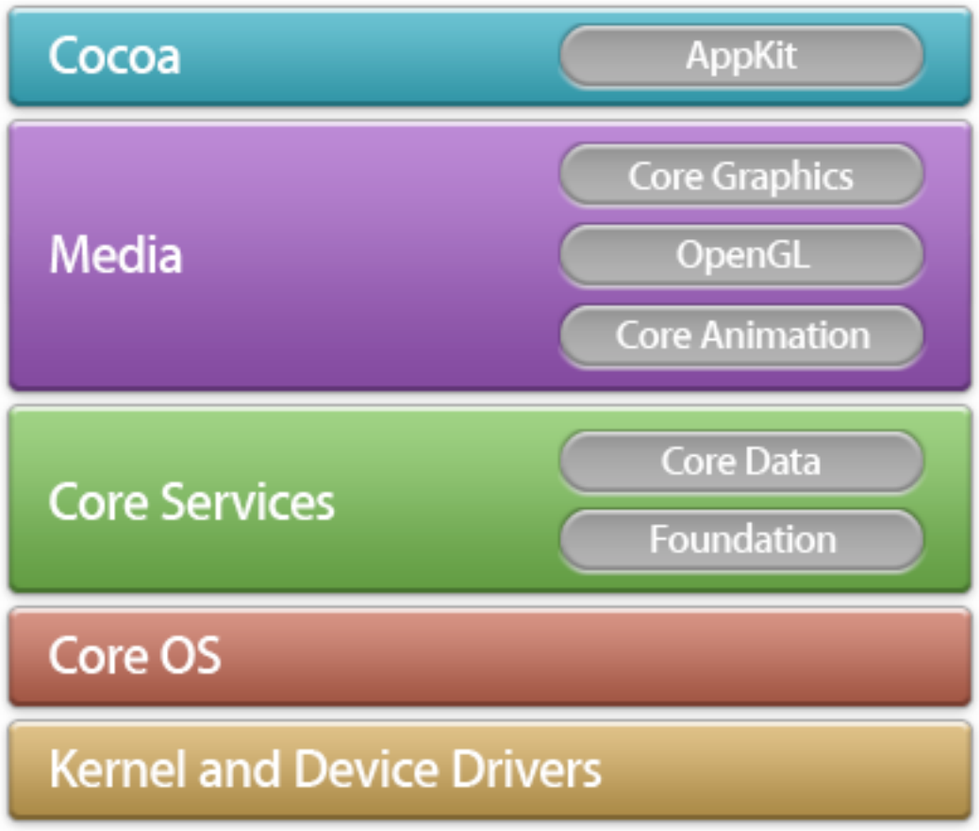
Frameworks

A framework

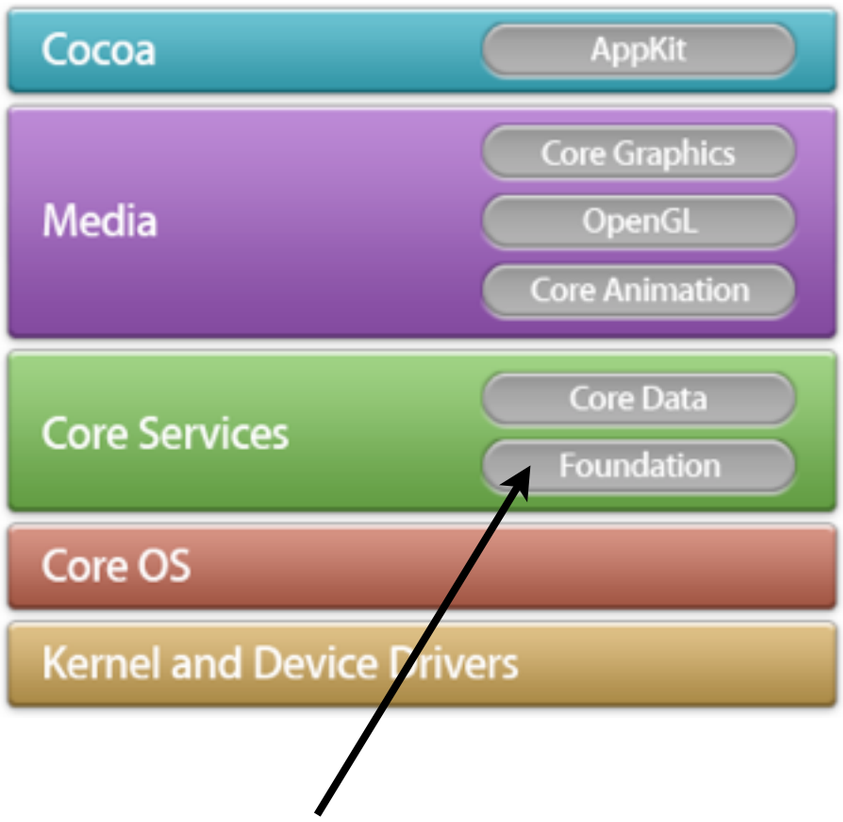
A framework is a collection of classes, methods, functions, and documentation logically grouped together to make developing programs easier.


Application code
Application code
Application code
Foundation code
Application code

Mac OS X structure

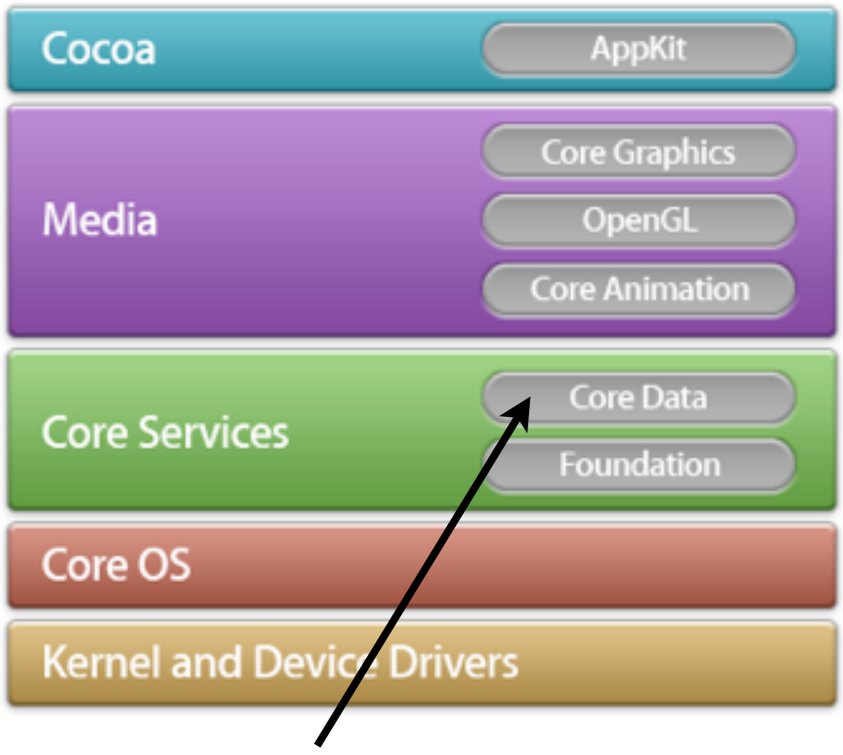


Foundation Framework



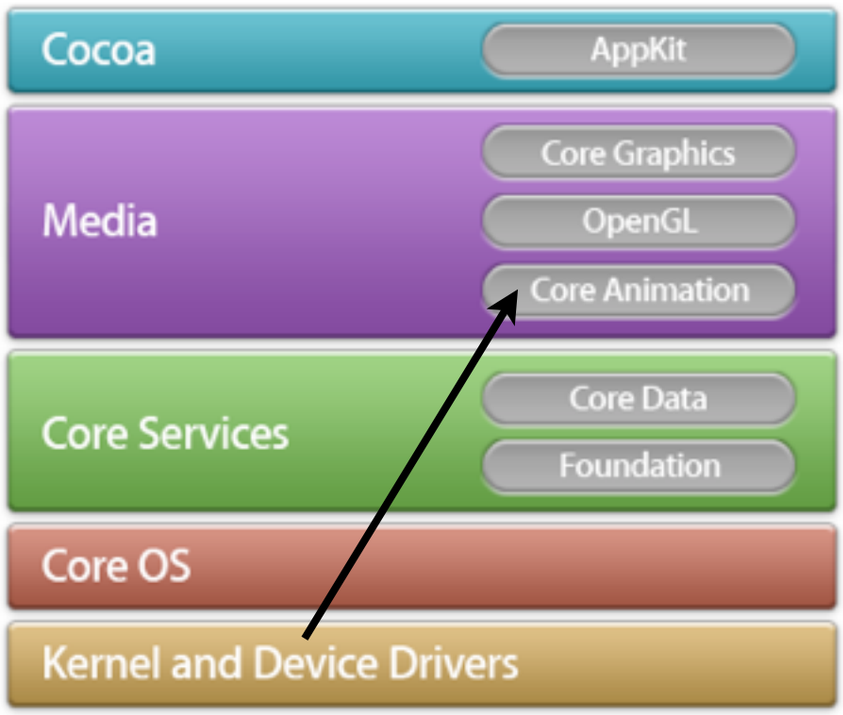
- Basic objects like numbers, strings, etc.
- Collections – arrays, sets, dictionaries, etc.
- Dates, times
- File system
- Storing objects
- Geometric data structures (points, rectangles, etc)

Core Data Framework



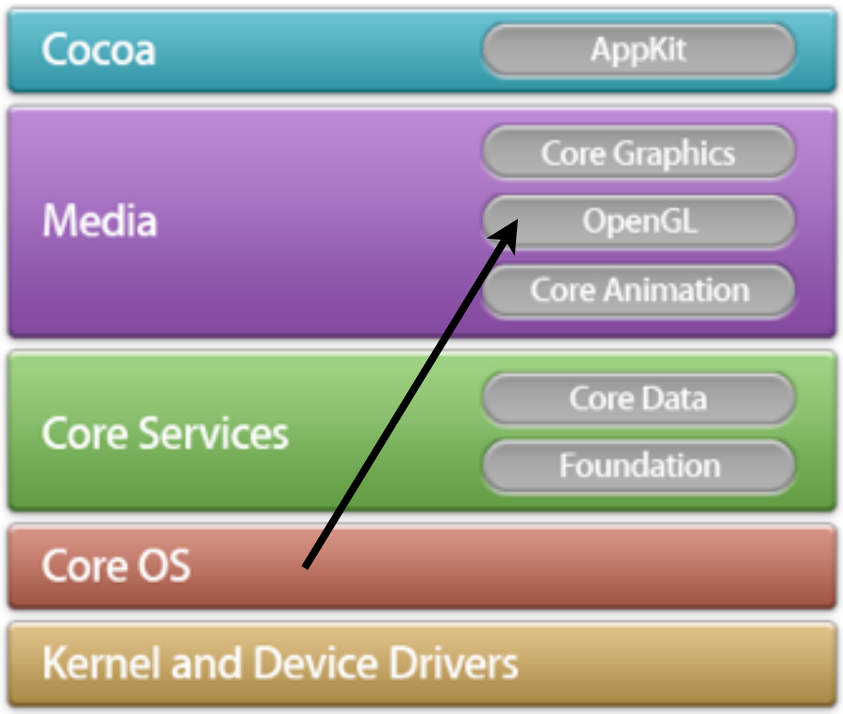
- Save and retrieve objects from storage
- Support basic undo/redo
- Validate property values automatically
- Filter, group, and organize data in memory
- Manage results in a table view with NSFetchedResultsController
- Support document-based apps

Core Animation



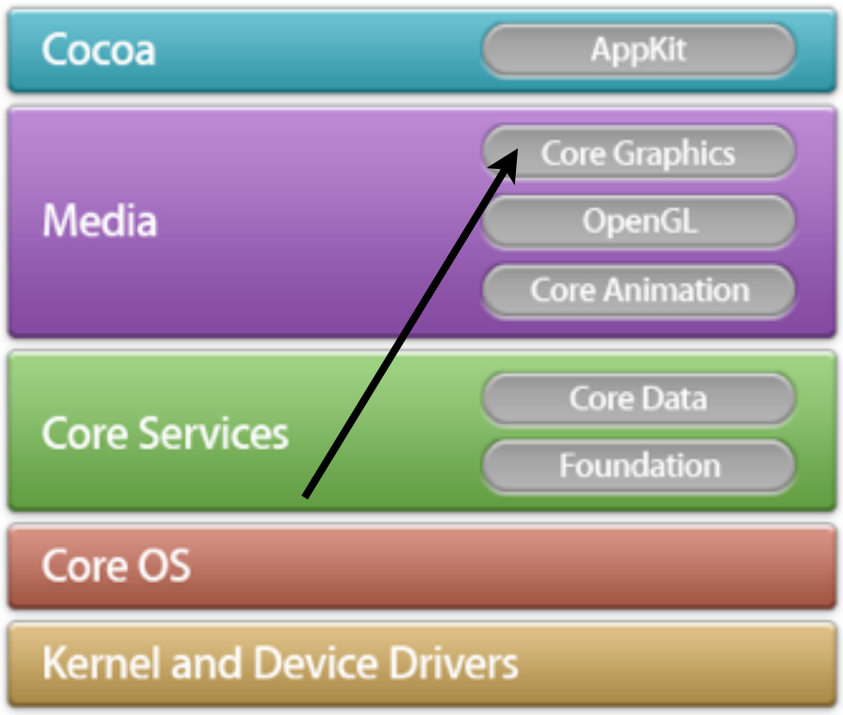
- Create custom animations
- Add timing functions to graphics
- Support key frame animation
- Specify graphical layout constraints
- Group multiple-layer changes into an atomic update

OpenGL Framework



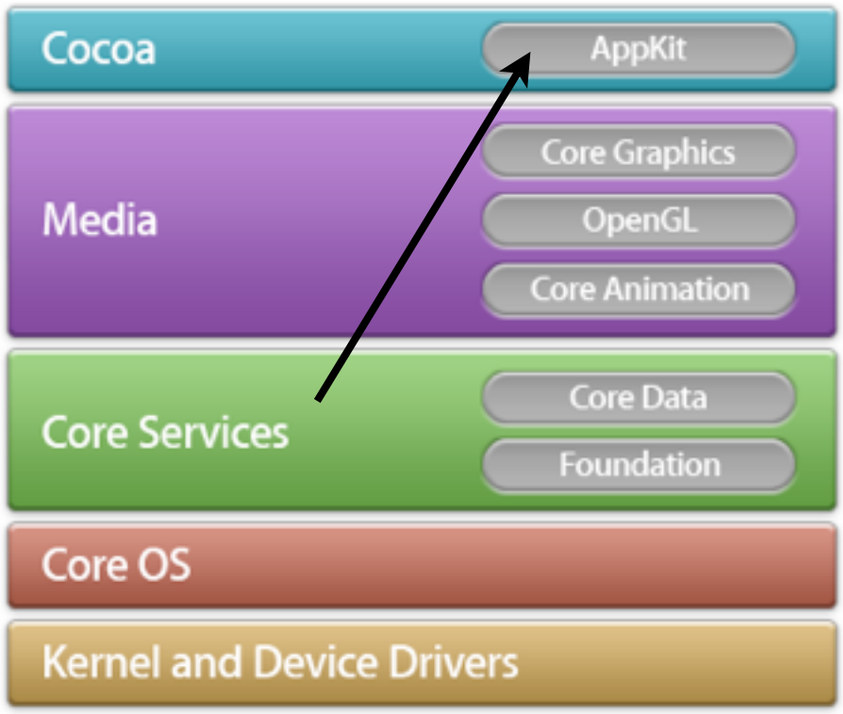
- Create 2D and 3D graphics
- Make more complex graphics such as data visualization, flight simulation, or video games
- Use multiple threads to process graphics data
- Access underlying graphics hardware

OpenGL Framework



- Make path-based drawings
- Use antialiased rendering
- Add gradients, images, and colors
- Use coordinate-space transformations

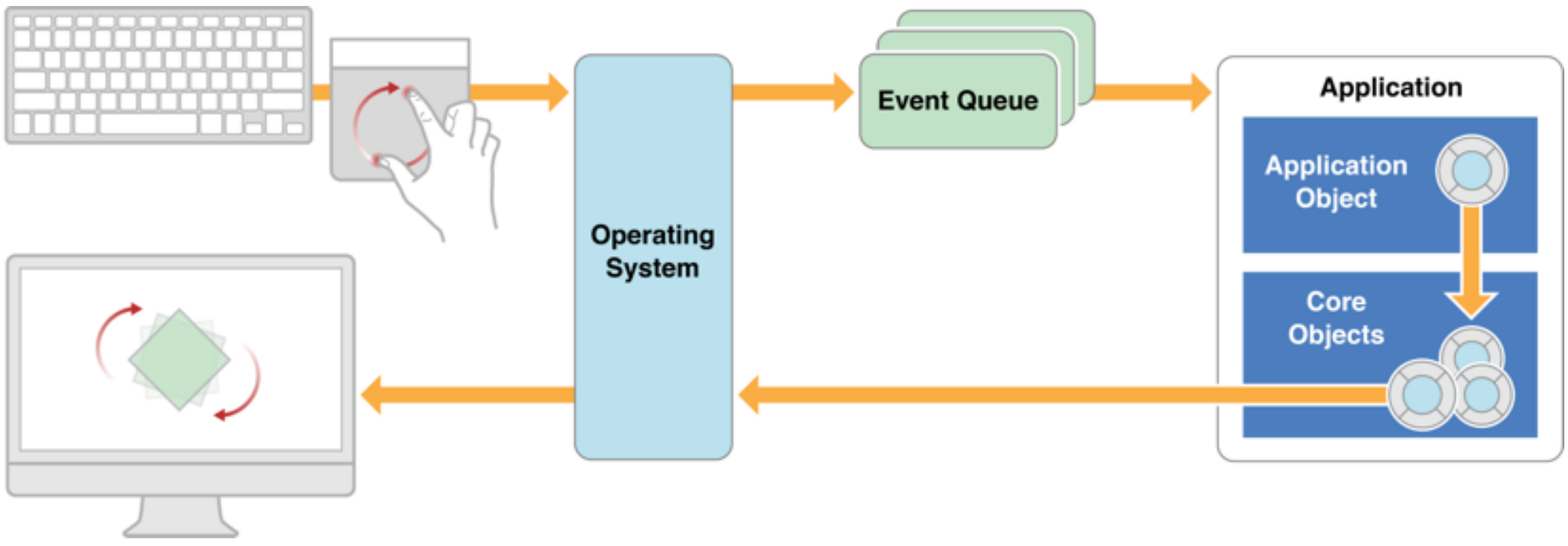
AppKit Framework



- Construct and manage your user interface
- Handle user events
- Present fonts, colours, and images
- Perform basic animation
- Support basic app features such as Spaces, Help Support, and multiple user accounts
- Create custom user interface elements

How apps work

- Draw the app
- Wait for events



Foundation documentation

- Full documentation in XCode
- Option+Click on a statement in code
- Quick help in Utility panel

Foundation: Numbers

```
NSNumber *myNumber, *floatNumber, *intNumber;
NSInteger myInt;

// integer value
intNumber = [NSNumber numberWithInteger: 100];
myInt = [intNumber integerValue];
NSLog (@"%li", (long) myInt);

// long value
myNumber = [NSNumber numberWithLong: 0xabcdef];
NSLog (@"%lx", [myNumber longValue]);

// char value
myNumber = [NSNumber numberWithChar: 'X'];
NSLog (@"%c", [myNumber charValue]);

// float value
floatNumber = [NSNumber numberWithFloat: 100.00];
NSLog (@"%g", [floatNumber floatValue]);

// double
myNumber = [NSNumber numberWithDouble: 12345e+15];
NSLog (@"%lg", [myNumber doubleValue]);
```

Foundation: Strings

```
@"This is a NSString"
```

This actually creates a NSConstantString, which is a subclass of NSString.

It's called an immutable object.

NSLog

```
NSString *str = @"Why u no..?";
NSLog (@"%@", str);
```

You can define what gets printed from your class using @ in NSLog by overriding description method.

Description method

```
-(NSString *) description {
    return [NSString stringWithFormat: @"%i/%i", numerator, denominator];
}
```

Let's see it in action (if we have time...)

Foundation: Arrays

- Elements of an array are not required to be of the same type
- There are immutable (NSArray) and mutable (NSMutableArray) arrays